Curtin University — Department of Computing

UNIX and C Programming (COMP1000)

Semester 2, 2015

Assignment

Due: Week 13 – Tuesday 27 October, 12:00 noon.

Weight: 20% of the unit mark.

Your task for this assignment is to create, in C89, a stand-alone spell-checker program. There is some pre-existing code – `check.c` and `check.h` – to help with this.

1 Documentation

You must thoroughly document your code using C comments (`/* ... */`).

For each function you define and each datatype you declare (e.g. using `struct` or `typedef`), place a comment immediately above it explaining its purpose, how it works, and how it relates to other functions and types. Collectively, these comments should explain your design. (They are worth substantial marks – see Section 7.)

2 Program Outline

Your program will deal with *three input files*: a text file to check (the "user file"), a dictionary file containing correct spellings, and a settings file. The words in the user file will be spell-checked, and any resulting corrections written back to that file.

Your program must do the following:

(a) Take one command-line parameter – the name of the user file to be spell-checked.

(b) Read the settings file "`spellrc`" and store its contents in a **struct**. The format of the settings file is described in Section 2.1. In brief, it contains the following information:

- The name of the dictionary file;
- Whether or not to auto-correct the spelling; and
- The maximum difference between a misspelled word and any suggested correction.

(c) Read the dictionary file (whose name is given inside the settings file), placing each dictionary word into a **linked list**. The dictionary file contains words sepa-

rated by line breaks. Once the whole dictionary file has been read, your program should copy the contents of the linked list to a **dynamically-allocated array**.

(d) Read the user file in the same fashion. For the sake of simplicity, assume that the file consists entirely of words separated by whitespace. (That is, you *do not* have to make special allowances for punctuation, though you can if you wish.)

(e) Invoke the pre-existing function `check()`, providing a callback (pointer to a function). The `check()` function is described in Section 2.2, and the callback function in Section 2.3.

(f) Write the user file array (whose contents may have been corrected) back to the user file. This obviously has to happen *after* the `check()` function has finished. For the sake of simplicity, you *do not* need to preserve the original formatting.

Note: you may assume that no word is longer than 50 characters, in both the dictionary and user file.

2.1 The settings file

The settings file must be called "`spellrc`" (no filename extension).

Each line in the file contains a name, then "`=`", then a value (separated by whitespace). The allowed setting names are: `dictionary`, `maxcorrection` and `autocorrect`. These settings may appear in any order, and can be repeated. If a particular setting name is repeated, the value of the last (bottom-most) occurrence must take effect.

For the `dictionary` setting, the value is the name of the dictionary file.

For `maxcorrection`, the value is an integer representing the maximum allowable difference between a misspelt word and any corrections to be found. (This is the "edit distance" between two words — the number of single-character changes it takes to transform one word into another.)

For `autocorrect`, the value is either "`yes`" or "`no`", indicating whether your program should automatically apply any corrections found, or ask the user first.

This is an example of a valid settings file:

```
maxcorrection = 2
dictionary = thedictionary.txt
autocorrect = no
```

Your program should store these settings in a struct for later use.

Note: if a setting is missing from the file, or there is an unrecognised setting name, or the file is otherwise not in the expected format, your program must issue an error message detailing the problem. When this happens, your program must not attempt to spell check anything.

2.2 The `check()` function

The `check()` function, provided in `check.h` and `check.c`, is defined as follows:

```
void check(char* text[], int textLength,
           char* dict[], int dictLength,
           int maxCorrection, ActionFunc action)
```

Your program must *use* this function to perform the actual spell-checking.

The parameters are as follows:

- `text` — an array of words to spell check (each word *must* be dynamically allocated);
- `textLength` — the number of words to spell check;
- `dict` — an array of words to use as the dictionary;
- `dictLength` — the number of dictionary words;
- `maxCorrection` — the maximum difference between misspelt words and their suggested corrections;
- `action` — a pointer to a function that will be called for each misspelt word.

The `check()` function checks the spelling of each word in `text` against the dictionary words in `dict`. If no exact match is found, it tries to find the closest matching word.

The `maxCorrection` parameter should be set according to the `maxcorrection` setting (from the settings file).

2.3 The `ActionFunc` callback

The `check()` function takes a function pointer of type `ActionFunc`, defined as follows:

```
typedef int (*ActionFunc)(char* word, char* suggestion);
```

You must write one or more functions matching this definition, and pass one to the `check()` function.

Your callback function will receive two parameters:

- `word` — a misspelt word;
- `suggestion` — a suggested correction, or `NULL` if the word is too badly misspelt.

Your callback function must make a decision on whether to accept the suggested correction or not. It should return `TRUE` if the correction is to be accepted (and the original word modified), or `FALSE` if the misspelling should be ignored.

This decision will be based on the `autocorrect` setting (from the settings file). If `autocorrect` is `yes`, then all words that *can* be corrected should be. Otherwise, the user should be presented with the original word and the corrected version, and asked to choose between them.

If `suggestion` is `NULL`, no correction is possible, irrespective of the `autocorrect` setting.

3 Makefile (or "How to Actually Get Marks, part 1")

You must create a makefile, and it must actually work. That is, the marker will type:

```
[user@pc]$ make
```

This is the only way the marker will attempt to compile your code. If your makefile does not work, then, according to the marking guide, *your code does not compile or run*. (The marker will delete all existing .o files and executable files beforehand.)

Your Makefile must be written by you, and *not* automatically generated. It must be structured properly in accordance with the lecture notes.

4 Testing (or "How to Actually Get Marks, part 2")

Your program must work on either:

- The lab machines in building 314, and/or
- One of the `saeshell0Xp.curtin.edu.au` machines (where "`X`" is 1, 2, 3 or 4).

To begin with, construct a small-scale test: dictionary and user files containing only a few words. As you begin to fix bugs, try larger test cases. You will find a full-scale dictionary file in the directory `/usr/share/dict/` (on Linux systems).

Use `valgrind` early and often! You *will lose marks* for failing to fix issues reported by `valgrind`, and your program could crash unpredictably anyway.

5 README.txt

Prepare a text file called `README.txt` (*not* using Word or any other word processor), that contains the following:

- A list of all files you're submitting and their purpose (if it's not obvious).
- A statement of how much of the assignment you completed; specifically:
  - How much of the required functionality you attempted to get working, and
  - How much *actually does* work, to the best of your knowledge.
- A list of issues reported by `valgrind`, if any, along with any other bugs or defects you know about, if any.
- A statement of which computer you tested your code on. This *must* be either:
  - One of the four `saeshell0Xp` machines, or
  - One of the building-314 lab machines. In this case, specify the room number (e.g. 314.220) and the "Service Tag" – a unique 7-character ID found on a black sticker on the top of the each machine. See next page.

Sticker on top of each lab machine.

6 Submission

You must submit the following electronically, via the assignment area on Blackboard, inside a single .zip or .tar.gz file (not .rar or other formats):

- Your `makefile` and `README.txt`.
- All .c and .h files (everything needed for the `make` command to work).

You are responsible for ensuring that your submission is correct and not corrupted. You may make multiple submissions, but only your newest submission will be marked. The late submission policy (see the Unit Outline) will be strictly enforced. A submission *1 second* late, according to Blackboard, will be considered *1 day* late. A submission *24 hours and 1 second* late will be considered *2 days* late, and so on.

You must also submit a completed, signed "**Declaration of Originality**" form to the lecturer or unit coordinator **in person** (*not* electronically). Look for blank forms at the counter on level 3, building 314, School of Electrical Engineering and Computing.

7 Mark Allocation

Here is a rough breakdown of how marks will be awarded. The percentages are of the total possible assignment mark:

10% – Using code comments, you have provided good, meaningful explanations of all the files, functions and data structures needed for your implementation.

10% – Your have followed good coding practices, and your code is well-structured, including being separated into various, appropriate .c and .h files.

20% – You have correctly implementated the required functionality, according to a visual inspection of your code by the marker.

20% – Your program compiles, runs and performs the required tasks. The marker will use test data, representative of all likely scenarios, to verify this. You will not have access to the marker's test data yourself.

You may lose this entire component of your mark by either (a) not having a working makefile, OR (b) failing to solve issues raised by `valgrind`.

40% – For the all practical signoffs put together (i.e. 5% per prac signoff).

8 Academic Misconduct – Plagiarism and Collusion

If you accept or copy code (or other material) from other people, websites, etc. and submit it, **you are guilty of plagiarism**, unless you correctly cite your source(s). Even if you extensively modify their code, it is still plagiarism.

Exchanging assignment solutions, or parts thereof, with other students is **collusion**.

Engaging in such activities may lead to a grade of ANN (Result Annulled Due to Academic Misconduct) being awarded for the unit, or other penalties. Serious or repeated offences may result in termination or expulsion.

You are expected to understand this at all times, across all your university studies, *with or without* warnings like this.

End of Assignment